Chapters 7-8-9-10
JavaScript Basics
(Control Flow-Functions-Arrays-Objects)

--- CONTROL FLOW ---

Boolean Logic
- == Equal to
- === Equal value and type
ex:
```
x = 5
x == "5" //true
x === "5" //false
```
ex:
```
var y = null
y == undefined //true
y === undefined //false
```
- '==' performs "type coersion". It tries to get them to same format and then compare the value that's why int 5 equals to string 5.
- It is strongly recommended to use '===' because it is much safer!
- Following exmaples will give an idea about non-reliablity of '==',
ex:
```
true == "1"  //true
0 == false  //true
null == undefined  //true
NaN == NaN  //false
```

Logical Operators
1. `&&` - AND
2. `||` - OR
3. `!` - NOT
- Values that aren't actually true or false, are still inherently 'truthy' or 'falsy' when evaluated in a boolean context,
```
!"hello" //false which means string is "truthy"
!!"hell" //true which also means string is "truthy"
!!"" //false
!!null //false
!!0 //false
!!NaN //false
!!-1 //true
```
- Falsy values:
`false`, `0`, `""`, `null`, `undefined`, `NaN`

- propmt always returns a String! So in order to convert it into number,
```
var age = Number(prompt("What is your age?"));
```

Conditionals
ex:
```
if(age < 18){
    console.log("I'm sorry. You are not old enough to pass!")
  }
  else if(age < 21){
    console.log("All right, pass but my eyes are on you!")
  }
  else {
    console.log("Do whatever you want...")
  }
```

- `typeof myNumber`
  Gives us the type of the variable "myNumber"

- DRY: Don't Repeat Yourself (x WET: Write Everything Twice)
  We want to keep our code as DRY as possible.

Loops
1. while
   ex:
```
var count = 1;
    while(count <= 10) {
      console.log("count is: " + count);
      count+=2;
    }
```
When we evaluate the example above in chrome console, we will see that it also shows us number 11 which is not expected. But actually it's not printing it out it just shows us that as it is the lastly evaluated value but not printed out because at the last loop of the while count is updated to 11. And to point this out, console just pust a light-reverse-directed arrow to the beginning of the line.
   ex:
```
var str = "hello";
    var count = 0;
    while(count < str.length) {
      console.log(str[count]);
      count++;
    }
```

- `str.indexOf("a");`
  Gives me the index of first 'a' occurs in the text.
- `str.indexOf("ali");`
  Gives me the index of first 'ali' occurs in the text.
- `str.indexOf("x");`
  Gives me '-1' if x does not exist

--- FUNCTIONS ---

- `for(init; condition; step) {}`

- Functions let us wrap bits of code up into reusabel packages. They are one of the building blocks of JS,
```
funciton doSomething() { //declaring:
 console.log("Hello World!");
  }
doSomething(); //calling:
```
If I just type the function name without parantheses and hit enter, console will give me function itself but not run it!
```
doSomething
```
With arguments,
```
function area(height, width) {
      console.log(height * width);
    }
area(9, 8) //72
```

- If I leave blank an argument whilce calling a function in JS, it will take it as undefined and not cause an error

- `clear()`
  Typing and hitting enter will clear the console

- Every function in JS returns something and if we don't specify it explicitly it will simply return 'undefined'. That's the reason why we've been seeing 'undefined' consistently in the console while we were printing out console.log messages with functions

```
- function capitalize(str) {
    return str.charAt(0).toUpperCase() + str.slice(1);
  }
  var city = "wien"; //wien
 var capital = capitalize(city); //Wien
```

- `str.charAt(index)` Takes the character at 'index' of the string 'str'

- `str.toUpperCase()` Takes the string 'str' (or a single char if given) and turns it into upper case

- `str.slice(2)` Slices the string 'str' and takes the part starting (by including the given index) from the index till the end of the string

- `return` Simply stops the execution of the function

- `if(typeof(str) === "number") {}` Checks the type of str that if it's a number or not

Different Syntaxes for Creating a Function

1. Function Declaration,

```
function capitalize(str) {
     return str.charAt(0).toUpperCase() + str.slice(1);
   }
```

2. Function Expression,

```
var capitalize = function(str) {
     return str.charAt(0).toUpperCase() + str.slice(1);
   }
```

3. The slight difference between the two is, when I assign something else to the variable at the 'Function Expression' then I lose my function as expected.
4. In 'Function Expression' syntax, function names can be anonymous (can be left blank)
5. More on these two types: (https://javascriptweblog.wordpress.com/2010/07/06/function-declarations-vs-function-expressions/)

- `return num % 2 === 0;` Returns true if the num is even, false otherwise

- `str.replace(/-/g, "_");` Replaces all "-" to "_"

Scope

- It is the context that code is executed in

ex:

```
var num = 1;
  function nev() {
    var num = 2;
  }
  console.log(num); //prints 1
```

ex:

```
var num = 1;
  function nev() {
    num = 2;
  }
  console.log(num); //prints 2
```

Higher Order Functions

1. Functions that take other functions as argument.
2. `setInterval(anotherFunc, intervalMs)`: Takes another function and time as ms as arguments and repeats that function at every ms that was given. The reason why we don't use parantheses here while passing the function, we are passing the function itself as the source-code to the `setInterval()` function and lets it to call that code and execute that stuff by itself,

```
setInterval(sing, 1000);
```

3. As soon as we type and hit enter it returns us a number and then starts executing the code at given intervals. That number is the key for us to stop the interval. Let's say it was '2', so to stop the interval by the function '`clearInterval()`' we type,

```
clearInterval(2);
```

4. We also can use an anonymous -not predefined- function within the `setInterval()` like,

```
setInterval(function(){
      console.log("This is an anonymous function.")
   }, 1000)
```

5. 1000 ms = 1 second

--- ARRAYS ---

- Ways of creating an array,

1. `var friends = [];` *//common*
2. `var friends = new Array()` *//uncommon*

- Syntax of an array,

```
var friends = ["Emma", "John", "Betty", "Dave"];
```

- Index numbers in JS starts from 0,

```
friends[0]; //"Emma"
```

- We can directly change an element at an index by reassigning,

```
friends[1]; //"John"
friends[1] = "Johnny";
friends[1] //"Johnny"
```

- We can add new data to an array,

```
friends[4] = "Julia";
  friends //["Emma", "John", "Betty", "Dave", "Julia"];
```

- If we add the new data to a further place,

```
friends[7] = "Chuck";
```

Then JS will fill the indexes between with "undefined",

```
friends; //["Emma", "John", "Betty", "Dave", "Julia", undefined, undefined, "Chuck"];
```

- Arrays can hold different types of data altogether. All individual items doesn't need to be of the same type,

```
var mixed = [1, "Anna", true, undefined, 52, "Türkei", null];
```

- Length property will give us the length of the array just like strings,

```
friends.length //8
```

Built-in Array Methods (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array#)

1. `push/pop`

We use '`push`' to add item to the end of the array. It also returns the updated length of the array right away, it can be stored in a variable for further use,

```
var abc = ["a", "b"];
   var.push("c") //3
abc //["a", "b", "c"]
```

We use 'pop' to remove the last item of the array. It also returns us that last element right away, it can be stored in a variable for further use,

```
abc.pop() //"c"
abc //["a", "b"]
```

2. unshift/shift

They are similar to 'push/pop' pair but just does the same thing at the beginning of the array.

So we use 'unshift' to add item to the beginning of the array, It also returns us the updated length of the array right away, it can be stored in a variable for further use,

```
var nums = [2, 3, 4];
var.unshift(1) //4
nums //[1, 2, 3, 4]
```

We use 'shift' to remove the first item of the array. It also returns us that first element right away, it can be stored in a variable for further use,

```
nums.shift() //1
nums //[2, 3, 4]
```

3. indexOf

It's used to find the index of the first occurence of an element in an array, it simply returns us that index and if the given element is not found within the array it returns '-1',

```
var myFriends = ["Ahmet", "Mehmet", "Hasan", "Ahmet"];
myFriends.indexOf("Mehmet"); //1
myFriends.indexOf("Ahmet"); //0 (not 3)
myFriends.indexOf("Hakan"); //-1
```

4. slice

It's used to copy portions or all of the array,

```
var colors = ["blue", "red", "green", "yellow", "black"];
var myColors = colors.slice(1,3); //it copies the [1,3) elements, so first index is included but the second is not!
myColors //["red", "green"]
var colorsCopy = colors.slice(); //copies all elements
colorsCopy //["blue", "red", "green", "yellow", "black"];
```

PS: The original array is not manipulated, it stays the same

EX:

1.
```
var numbers = [11, 22, 33, 44];
console.log(numbers[numbers.legth]) //prints undefined
```
2.
```
var names = [ ["Ali", "Veli"], ["Ahmet", "Mehmet"], ["Hasan", "Hüseyin"] ];
console.log(names[1]) //["Ahmet", "Mehmet"]
console.log(names[2][0]) //"Hasan"
```

Note by Lecturer,

Chrome browser behaves a little strangely when using alert, prompt, or confirm functions. It doesn't display the HTML on the page until after the popup has been closed. This is problematic since our HTML contains instructions for the user to be able to use the app we're building.

You can avoid this by wrapping your JS code in the following setTimeout function:

```
window.setTimeout(function() {
    // put all of your JS code from the lecture here
  }, 500);
```

This gives the HTML a half second to load before running the JS, which circumvents the issue of the prompt function blocking the HTML from loading right away.

This is not something you would have to deal with in the real world as prompt, alert, and confirm functions are almost never used and when they are it's typically not on page load.

You'll also learn jQuery in latter sections which has a cool `$('document').ready()` function that you could use in place of the `window.setTimeout` workaround mentioned above.

Array Iteration

1. Classical for loop,

```
var colors = ["rot", "grün", "gelb"];
   for(var i = 0; i < colors.length; i++){
     console.log(colors[i]);
   }
```

2. `.forEach` loop,

```
arr.forEach(someFunction)
```

ex:

```
var colors = ["rot", "grün", "gelb"];
   colors.forEach(function(color){
  //color is a placeholder, it can be renamed as wished
   console.log(color);
   });
```

Here if we don't give an argument to the anonymous function inside, it will repeat the things inside the function as many times as the number of elements in the array. So in conclusion, that argument hold that specific item in the array in every iteration and gives us the chance to use it as we want. Lastly we can use either a pre-defined function or an anonymous function.

Note by Lecturer,

`.forEach` takes a callback function, that callback function is expected to have at least 1, but up to 3, arguments. This is how .forEach was designed.

The arguments are in a specific order:

- The first one represents each element in the array (per loop iteration) that `.forEach` was called on.
- The second represents the index of said element.
- The third represents the array that `.forEach` was called on (it will be the same for every iteration of the loop).

ex:

```
[1,2,3].forEach(function(el, i, arr) {
    console.log(el, i, arr);
  });
```

- Splice method is used to delete a specific item with the given index.

```
.splice(index, howManyItemsToDelete)
```

- Be careful when using `.forEach` because when we use return inside of it, we just return from the anonymous function not from the `.forEach`!

- As soon as I call the function with parantheses at the end, then the function is executed, otherwise not.

- To make myForEach function like `arr.myForEach`, we write,

```
Array.prototype.myForEach = function(func) {
  //loop through the array
  for(var i = 0; i < this.length; i++){
   //call func for each item
   func(this[i]);
   }
  }
```

--- OBJECTS ---

- It consists of key-value pairs. They are not part of an ordering.

- Objects can hold all sorts of data, nums, strings, arrays even other objects...

- An object syntax seems like the following,

```
var person = {
   name: "Cindy",
   age: 32,
   city: "Köln"
 };
```

- There are 3 ways of creating an object,

2. Create'em all at once, just like above,

```
var person = {
    name: "Travis",
    age: 22,
    city: "Linz"
  };
```

1. Make an empty object then add to it,

```
var person = {}
  person.name = "Travis";
  person.age = 22;
  person.city = "Linz";
```

3. Make the empty object with a different notation,

```
var person = new.Object();
  person.name = "Travis";
  person.age = 22;
  person.city = "Linz";
```

- Updating the properties,

```
person["age"] += 1;
 person.city = "Berlin";
```

- To retrieve data from objects there are two ways:

`1. person["name"]` *//bracket notation*
`2. person.name` *//dot notation*

- We cannot use dot notation if the property starts with a number,

`someObject.1blah` *//INVALID*
`someObject["1blah"]` *//VALID*

- We cannot use dot notation if the property has spaces in its name,

`someObject.blah blah` *//INVALID*
someObject["blah blah"] *//VALID*

- We can lookup using a variable with bracket notation,

```
var str = "name";
```

`someObject.str` *//doesn't look for "name"*
`someObject[str]` *//does evaluate str and looks for "name"*

- In some languages 'objects' are called as 'dictionaries' thanks to the key-value pairs.

- Actually arrays are a special type of objects that has the keys only as numbers.

- To add a new key-value pair to an object,

```
person.surname = "Odd";
```

- An array of object is like the following,

```
var posts = [
    {
      title: "Dogs are awesome",
      author: "Ryan",
      comments: ["Yeah I think so", "Nope!"]
    },
    {
      title: "Cats are awesome",
      author: "Bob",
      comments: ["Not really", "Of course..."]
    }
  ]
  posts[1].comments[0]; //"Not really"
  var myObject = {
    friends: [
      {name: "Ali"},
      {name: "Veli"},
      {name: "Ahmet"}
    ],
    color: "blue",
    isEvil: false
  }
  myObject.friends[0].name; //"Ali"
```

- We can add functions into the objects as well as properties and when we do it we call these functions as "methods" after this point,

```
var obj = {
  name: "Mustafa",
  age: 21,
  isCool: false,
  friends: ["merve", "osman"],
  add: function(x,y){
    return x+y;
  }
}
```

To call methods,

```
obj.add(5,4);
```

It is just like "`console.log()`". "`console`" is an object and "`log()`" is a method inside of it

- Both to avoid namespace collisions and keeping the code organized we can do sth. like following,

```
var dogSpace = {};
  dogSpace.speak = function(){
    return "WOOF!";
  }
  var catSpace = {};
  catSpace.speak = function(){
    return "MEOW!";
  }
  dogSpace.speak(); //"WOOF!"
  catSpace.speak(); //"MEOW!"
```

This is used i.e. with delete functions for posts or comments or users in real world. They are both delete functions but refer to the different objects and do slightly different tasks.

This is also used in libraries. When we import a library, simply to use its methods we first refer to the library itself which is actually an object or in other words a namespace.

- Keyword "this" is used within the methods actually,

```
  var comments = {};
  comments.data = ["Good job!", "Thanks", "Congrats."];
  comments.print = function(){
    this.data.forEach(function(el){ //here "this" refers to the "comments" object!
   console.log(el);
    });
  };
  comments.print(); //Good job! \n Thanks \n Congrats.
```